SAND99-0022C

Spiraling Edge: Fast Surface Reconstruction from Partially Organized Sample Points

Patricia Crossno
Sandia National Laboratories

Edward Angel
University of New Mexico

ABSTRACT

Many applications produce three-dimensional points that must be further processed to generate a surface. Surface reconstruction algorithms that start with a set of unorganized points are extremely time-consuming. Often, however, points are generated such that there is additional information available to the reconstruction algorithm. We present a specialized algorithm for surface reconstruction that is three orders of magnitude faster than algorithms for the general case. In addition to sample point locations, our algorithm starts with normal information and knowledge of each point's neighbors. Our algorithm produces a localized approximation to the surface by creating a star-shaped triangulation between a point and a subset of its nearest neighbors. This surface patch is extended by locally triangulating each of the points along the edge of the patch. As each edge point is triangulated, it is removed from the edge and new edge points along the patch's edge are inserted in its place. The updated edge spirals out over the surface until the edge encounters a surface boundary and stops growing in that direction, or until the edge reduces to a small hole that fills itself in.

1 INTRODUCTION

In recent years, the reconstruction of surfaces from sets of unorganized sample points in $\mathbf{R}^3$ has been an active area of research [1][3][5][6]. The published results of this research focus on solving the general case where no additional information beyond the location of the surface points is known. Although the methods developed for the general case can be used to construct surfaces from sample points where some organization already exists, the time required to construct a surface of significant size is unacceptably long. Times reported for constructing surfaces with twenty thousand points range between 10 and 20 minutes of CPU time, depending on the method used [1][3][5][6]. However, point sets often have some structure that can be used to speedup up the process. Consequently, we developed the Spiraling Edge (SE) algorithm presented in this paper.

The SE algorithm was developed to generate a surface from particle positions after the particles have finished distributing themselves over an isosurface in a volume data set [2]. In addition to the particle or point locations, the algorithm uses a surface normal at each point, a point type designation (*interior*, *boundary*, or *corner*), and for each point a list of neighboring points ordered by distance. Given these inputs, the SE algorithm generates a triangulation that approximates the surface much more rapidly than previous general case methods. Our algorithm requires only 2.3 seconds to reconstruct a surface from a set of over forty thousand points. In addition, the SE Algorithm does not assume that the points form a single connected surface.

2 RELATED WORK

In 1992, Hoppe et al. presented a surface reconstruction algorithm that starts with a set of unorganized points, and generates a triangulation that approximates the unknown underlying surface [5]. The algorithm does not exploit any outside information about the nature of the surface. Hoppe defines a surface to be connected, so he does make the assumption that there are not multiple disconnected surface regions represented by the point set. Timings on a 20 MIPS workstation for data sets ranging from 1000 to 21,740 points were between 19 and 2,135 seconds.

Edelsbrunner and Mücke published their **α-shapes** work in 1994 [3]. For a point set P in three-dimensional space, α-shapes are a family of shapes that are a generalization of the convex hull. Setting the value of α between infinity and zero generates different shapes. When α is set to infinity, the α-shape is the convex hull of P. As α decreases, the α-shape shrinks; cavities and holes may appear. Unlike Hoppe's work, an α-shape is not required to be a single connected surface. The algorithm for constructing α-shapes requires finding the three-dimensional Delaunay triangulation of the point set. Edelsbrunner and Mücke generalized a two-dimensional edge-flipping algorithm to three dimensions. In three dimensions, the Delaunay triangulation is actually a tetrahedralization of the convex hull of the points. The additional constraint of the α-shape sphere then removes a subset of the tetrahedral faces, or triangles, from the Delaunay triangulation. The timings of their incremental-flip algorithm on a 50 MHz MIPS R4000 ranged between 8.97 and 2,096.14 seconds for point sets ranging in size from 318 to 15,000 points.

There are several problems with using α-shapes to do surface reconstruction. In objects with holes, it requires some experimentation to find an α-value that produces the appropriate surface. In some cases, there is no α-value that produces the desired surface. Either the surface is webbed in areas where there should not be a surface, or holes are beginning to appear in what should be a solid surface. Teichmann and Capps overcame these limitations of the α-shapes algorithm in 1998 with anisotropic density-scaling [6]. However, their algorithm also requires intensive calculations to compute the three-dimensional Delaunay triangulation of the point set.

In 1998, Amenta, Bern, and Kamvysselis presented a surface reconstruction algorithm based on the three-dimensional Voronoi diagram and Delaunay triangulation [1]. The algorithm inputs unorganized sample points and outputs a set of triangles, which they call the *crust* of the sample points. The vertices of the crust triangles are all sample points, so the algorithm interpolates the sample points instead of approximating them as Hoppe's algorithm does. The crust connects only adjacent sample points on the surface, assuming that the surface is smooth and sufficiently sampled.

The algorithm has difficulties with sharp edges due to the difficulty in placing both the poles away from the surface. Amenta, Bern, and Kamvysselis recommend some heuristics for choosing the second pole, but they believe that cases could be constructed that would result in topologically incorrect surfaces regardless of the sampling density. They used an SGI Onyx with 512 megabytes of memory to test the running time of their algorithm against several data sets. The data sets ranged between 939 and 35,947 points, while the running times went from 2 to 23 minutes. The amount of time required to do a surface reconstruction is in the 15-minute range for an example with 20,021 points. Their paper states that the running time is dominated by the time needed to compute the Delaunay

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

triangulation, which they use to generate the Voronoi vertices of the point set.

3 OVERVIEW

The starting point for the SE algorithm is a set of points generated by an isosurface algorithm. As part of the process of obtaining these points, information in addition to their locations also is generated. For the particle system approach to isosurfaces, this extra information consists of an estimated normal at each point, a list of each point's neighbors, and a classification of each point as an *interior* point, a *boundary* point, or a *corner* point. The latter distinctions are used in the particle algorithm to prevent particles from escaping the region in which the data are defined.

Rather than using a global algorithm to create tetrahedra, we work locally to create a nearly planar triangulation between a point and its nearest neighbors. Using a point as the center of a ring of its neighbors, such as point 1 in Figure 1, the question becomes: which neighbors are close enough and provide a good enough aspect ratio so as to remain in the triangulation of the center point? Those neighbors that fail these criteria, such as neighbors 8, 9, 10, 11, 14 and 18, are removed from the ring. The remaining neighbors are triangulated along with the center point. The triangulation consists of placing edges between the center point and each of the remaining ring of neighbors, as well as placing edges between each of the adjacent neighbors on the ring. This process creates a wheel-like triangulation with the center point as the pivot and the ring of neighbors along the rim.

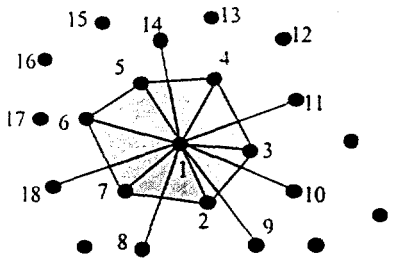

Figure 1: Central point with ring of neighboring points.

On a more global scale, our approach is to initialize an edge ring with a point and its nearest neighbor. After the algorithm triangulates a ring of points around the first point, it removes the encircled point from the edge ring and adds the new points used in the triangulation to the edge ring in its place. The edge ring traversal proceeds in counterclockwise order around the normal of the first point, thus providing an orientation for the triangulation that is consistent across the surface. The algorithm then spirals around the edge ring, triangulating each of the points along the ring in turn.

The algorithm assumes that except at points with *boundary* type designations the surface is closed and there are no non-manifold structures. The algorithm also assumes that there is only one point at any particular location. The algorithm grows a surface from a single point until one of two things happens. Either the surface encounters points that are surface boundaries and stops growing in that direction or the surface fills until the edge ring reduces to a small hole that fills itself in. Consequently, *boundary* points require special treatment.

Each time the edge ring becomes empty, a disconnected surface component has finished triangulation. If all of the points have been used, triangulation is complete. Otherwise, the algorithm reinitializes the edge ring with a new point and its nearest neighbor and continues.

Here is a pseudo-code description of the main steps in the SE algorithm:

```
Clear All Point's Status Flags to UNUSED
Repeat until All Points are USED
    Initialize Edge Ring
    While Edge Ring Not Empty
        If Point on Boundary
                Triangulate Vertex Fan
        Else Triangulate Vertex Ring
```

4 INITIALIZATION

The algorithm expects points to be classified as *interior*, *boundary* or *corner* types. These designations arise as part of the isosurface algorithm that generated the points and refer to where the points lie relative to the region over which the data are defined. The *boundary* points are further subdivided into one (or more) of six different subtypes, *positive x boundary*, *positive y boundary*, *positive z boundary*, *negative x boundary*, *negative y boundary*, and *negative z boundary*. We represent each of the *boundary* types by setting a unique bit within the point's type field. Any point that falls into two or more of these subtypes also sets the *corner* bit to speed up type checking operations. *Interior* points are those with no bits set.

Each point has a status flag that has three states: UNUSED, USED, and DONE. Before triangulation the flags are all cleared to UNUSED. Once a point has been used in a triangulation, its status flag is set to USED, which corresponds to a point being added to the edge ring. Because there are circumstances where the edge ring can include a point more than once (more will be said about this case later), the status flag also acts as an instance counter so that any value of the flag greater than 0 signals the USED state.

When a point is removed from the edge ring, its status flag is decremented. If the status flag is equal to 1 prior to a decrement, representing a single entry in the edge ring, the status flag is set to DONE instead of being decremented (the value of the DONE flag is −1). A DONE status indicates that a point has been entirely surrounded by a ring of triangles. Consequently, even if a DONE point has neighbor links to points that are not DONE, it cannot be a candidate in their triangulations. This situation can occur when neighbor links connect points that do not share a Voronoi face or edge.

As the flags are being cleared at startup, a pointer is initialized to the first point. This pointer is used later by `Initialize Edge Ring` to find a starting point. Each time `Initialize Edge Ring` is called, this pointer is updated to point to the current starting point. In that way, later searches for a new starting point will not have to repeatedly examine DONE points. Once the pointer advances past the last point, the algorithm knows that all points must be DONE because it calls this routine only when the edge ring is empty.

The edge ring is actually a doubly linked ring of elements containing pointers to the points. The edges are represented implicitly by the links between the points in the ring; consequently the ring needs to have at least two points. To initialize the edge ring, the algorithm inserts the first point that it finds that is not set to DONE, and the point's nearest neighbor that is not DONE. The algorithm begins the edge ring traversal with the first point.

Sometimes the algorithm encounters points that either do not have any neighbors or whose neighbors are all DONE. In the first case, these points represent such a tiny surface area that only a

single point will fit in that space. We view these points as noise and eliminate them. In the second case, the algorithm has rejected the point in the triangulations of all its neighbors and it has been left stranded. Although we prefer to use all of the points in representing the surface, the cost of locating the pertinent mesh region and re-triangulating that section to include the point outweighs the benefits of including it. So in both of these cases, the algorithm marks these points as DONE and it continues the search for a point to initialize the edge ring.

5 TRAVERSING THE EDGE RING WITH INTERIOR POINTS

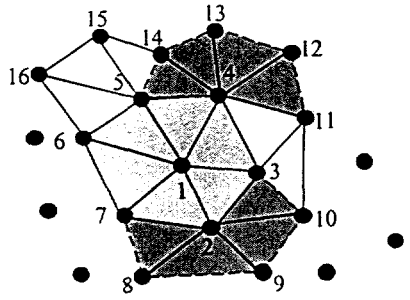

Figure 2: Spiraling triangulation of points on the edge ring.

Using Figure 2, we will step through several iterations of how the edge ring is manipulated when it consists exclusively of *interior* points. The normals of the points are assumed to be pointing out of the page. The edge ring is initially just point 1 and its nearest neighbor, point 2, with point 1 as the first point to be triangulated. After the algorithm triangulates the ring of points around point 1, it removes point 1 from the edge ring and adds points 3 through 7 to the edge ring. The algorithm then advances to the next point on the edge ring, point 2, and triangulates the section of point 2's ring of neighbors that has not yet been triangulated as part of point 1's triangulation. This sub-ring is shown in darker gray in the figure. The algorithm replaces point 2 on the edge ring with points 8, 9, and 10. Advancing to point 3, the algorithm triangulates point 3 (whose triangulation is shown in white) and replaces point 3 on the edge ring with point 11.

Continuing in this manner, the algorithm triangulates points 4 and 5, leaving an edge ring consisting of points 6 through 16, with point 6 as the next point to be triangulated. Each successive triangulation step is shown as a different color. Note that other than the original ring of triangles around point 1, all of the succeeding triangulations produce a fan of triangles that begins and ends with vertices on the edge ring.

6 TRIANGULATING AN INTERIOR POINT

The SE algorithm starts with three points: the center or pivot point, the point that precedes the pivot point on the edge ring (referred to as the first point), and the point that succeeds the pivot point on the edge ring (referred to as the last point). When the edge ring only consists of two points, as it does initially, the first and last points are the same.

6.1 Creating the Neighbor Ring

The normalized vector between the pivot point, **p**, and the **first** point is a reference vector where the angle is zero. We refer to this reference vector as **z**. Using **z**, the algorithm sorts all of the neighboring points by angle into a ring around the pivot point. If two neighboring points are found to have the same angle, the point closest to **p** is included in the ring and the other point is eliminated from consideration.

The algorithm uses the angle between **z** and the vector between the pivot point and the **last** point as the maximum allowable angle. When the **first** and **last** points are the same, the ending angle is 2π. The ending angle prevents including neighboring points in the ring that would require the triangulation to overlap regions that are already triangulated. In Figure 3, points **t** and **p** are linked as neighbors, but because the angle formed by **t** and **z** is greater than that formed by **last** and **z**, **t** is not included in the neighbor ring.

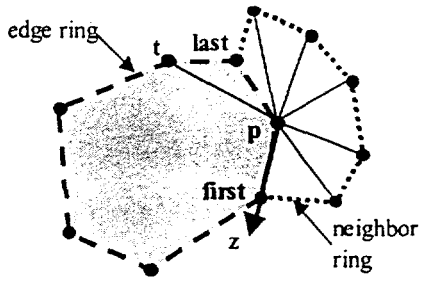

Figure 3: Using max angle to remove neighbor points from ring.

To calculate the angle, the algorithm does the following. First, it takes the dot product of the vector **z** and the normalized vector between the point **p** and a neighboring point, which we will call **v**, to obtain the cosine of the angle between them. Because the cosine is symmetric about π, the algorithm needs some way to differentiate which side of π the cosine lies in. Hence, we construct a plane through the pivot point and the first point by taking the cross product of **z** and **p**'s normals, producing the normal of the plane slicing the ring at 180 degrees. This vector, **n**, is normalized and the dot product of **n** and **v** is compared to zero. If $\mathbf{n} \cdot \mathbf{v} \geq 0$, then θ is set to the arccosine of $\mathbf{z} \cdot \mathbf{v}$, otherwise θ is set to 2π minus the arccosine of $\mathbf{z} \cdot \mathbf{v}$.

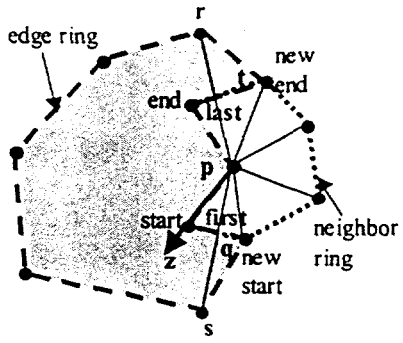

Figure 4: Changing the start and end points of the neighbor ring.

There are situations that arise that require the algorithm to change the starting and/or ending angle that it established using **first** and **last**. An example where both must be changed is presented in Figure 4. As in the previous discussion, the neighbor ring for **p** is initialized to contain only the two points **first** and **last**, which are the two adjacent points on the edge ring. The algorithm keeps pointers to the points whose angles represent the starting and ending angles within the neighbor ring. We call these pointers **start** and **end** and they initially point to **first** and **last**. The algorithm then looks at the two points along the edge ring to

either side of **first** and **last**, **q** and **t**, respectively, to determine if the edge ring is convex or concave at those two points. In this case both points are concave.

If as the algorithm adds in **q**, it finds that **q** is already on the edge ring, **q** is adjacent to the point pointed to by **start**, and the edge ring is concave at this point. The algorithm then advances the start pointer to point to **q**. The algorithm keeps **z** as the reference vector, but now the beginning angle is greater than zero. Similarly, the algorithm moves the end pointer when it adds **t** to the neighbor ring. Later, when the algorithm considers adding **s** and **r** to the neighbor ring, they are excluded because they fall outside the angle limits formed by the new start and end points. If **s** and **r** had already been included in the neighbor ring, the algorithm would remove them along with any other points in the neighbor ring between start and **q**.

In sorting the ring of neighbor points, we have made the assumption that they are in a nearly planar region. However, this assumption is not always true. We use the convex labeling of the start and end points to eliminate points from the neighbor ring which have passed the angle test, but which have been sorted incorrectly due to the three-dimensional distortions in a highly curved region.

An example where the angle sort fails is shown in Figure 5. The neighbor ring is shown as a dotted line that overlaps the edge ring between 1 and 2, and 3 and 4. The neighbor ring incorrectly includes point 2, which has been drawn slightly above point 1 for emphasis, but which is really level with **p** and point 1 and further back into the page. Point 3 is also further back into the page, while point 4 is closer to the viewer. The convexity test eliminates point 2 from the neighbor ring because point 2 is the next point on the edge list after the point pointed to by **start** (point 1) and 1 is convex.

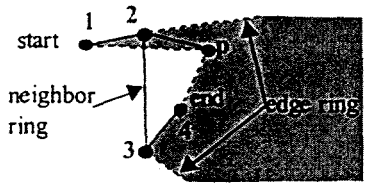

Figure 5: Convexity test needed to resolve angle sort failure.

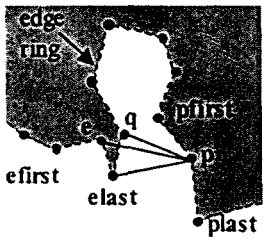

Figure 6: Overlap due to failure of angle test to exclude point **e**.

The convexity test only applies to points in the edge ring that are adjacent to the points pointed to by start or end since beyond this points on the edge ring could be on the other side of a hole being filled and could be legitimately included. On the other hand, there is the possibility of overlapping triangles caused by the inclusion of points on the edge ring that are in a configuration such as that in Figure 6. For point **p**, point **e** passes the angle test for the range of angles between **pfirst** and **plast**. However, if **e** is included in **p**'s neighbor ring, the resulting triangulation will overlap the region already triangulated. Consequently, points with a USED status are given an additional check before being added to the neighbor ring.

In the example shown in Figure 6, the edge point **e** is treated as the pivot point and the angles formed by the preceding and succeeding points on the edge list (**elast** and **efirst**) are calculated. Then the angle of **p** with respect to these beginning and ending angles for **e** is found. If **p** could be included in **e**'s hypothetical neighbor ring, then **e** is included in **p**'s neighbor ring. In this case, **p** could not be included, so **e** is eliminated from consideration for **p**'s neighbor ring.

6.2 Filling Holes

Once all of the neighboring points have been evaluated for inclusion in the neighbor ring, the algorithm must evaluate the points it has chosen for possible removal. First the algorithm checks for the special case of filling a hole formed by **p**, **first**, and **last**. Once the algorithm has used the tests already described to exclude unwanted edge ring neighbors from the neighbor ring, there are two possible types of neighbors remaining. These are UNUSED points inside the hole and UNUSED points outside the hole.

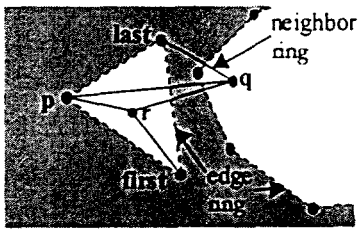

Figure 7: Eliminating neighbors outside the hole.

An example of this configuration is shown in Figure 7 where the neighbor ring consists of **first**, **r**, **q**, and **last**. The goal is to keep **r** and remove **q**. The algorithm constructs a plane by taking the cross product of the vector between **first** and **last** and the average normal of **p**, **first** and **last**. Only those points that are on the same side of the plane as **p** are kept in the neighbor ring. The other points, such as **q**, are removed from the neighbor ring.

Note that the algorithm will not recognize a hole formed by more than three points. If the example in Figure 7 contained an additional point on the edge list forming the hole, point **q** would be kept in the neighbor ring and produce an overlap in the triangulation. Although checks for larger holes could be made, it would be a case of diminishing returns. Since the distance between **p** and any of its neighbors is limited, the likelihood of **p** being linked to points on the other side of a triangulated strip decreases with the size of hole. However, there are still circumstances where overlaps occur.

6.3 Removing Neighbors from the Ring

The algorithm now evaluates each point in the neighbor ring for possible removal. The algorithm calculates the center of the circumcircle passing through the pivot point and the points on either side of the point being evaluated using a technique described in Graphics Gems IV [4]. Figure 8 presents an example evaluating point **q** for removal from the neighbor ring. A circumcircle is formed using points **p**, **s**, and **t**. The algorithm treats the circumcircle as though it were at the equator of a corresponding circumsphere. The center of the circumcircle can be used as the center of the circumsphere. The algorithm finds the distance **d** in three-dimensions between the center and **q**, and compares this to the radius of the circumcircle **r**. If **q** falls inside

the circumsphere or its removal would create an angle difference between **s** and **t** of more than 120 degrees, **q** must be kept. Otherwise, **q** is removed from the neighbor ring. In this case, **q** is removed and the evaluation moves on to point **s**.

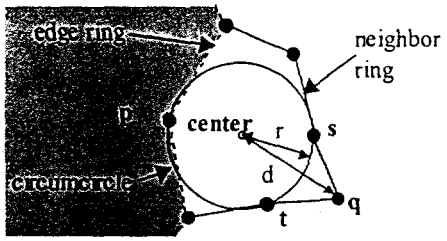

Figure 8: Evaluating neighbor ring points for removal.

6.4 Triangulating the Ring

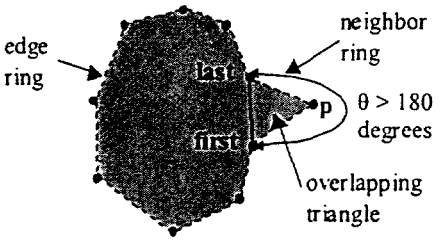

Figure 9: Testing for an overlapping triangle versus a hole.

Once the entire neighbor ring has been evaluated, one final check must be made before triangulating the remaining points. As is shown in Figure 9, if the neighbor ring consists of only two points and the angle they form around the pivot point is greater than 180 degrees, the resulting triangle will overlap existing triangles. In this case, the neighbor ring will be discarded and the routine **`Triangulate Vertex Ring`** will return without further action. The pivot point will advance to the next point on the edge ring and **p** will be triangulated indirectly by triangulating the points adjacent to it on the edge ring. The occurrence of this configuration indicates that **p** is insufficiently linked to its neighbors. This situation can arise when the neighboring points lie just outside the search range for **p**. If the angle between **first** and **last** is less than 180 degrees, a two-point neighbor ring represents a hole that is being filled and triangulation may proceed.

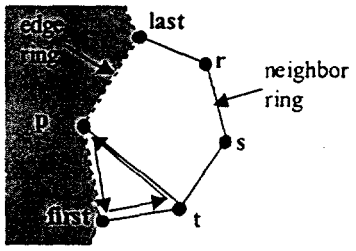

Figure 10: Creating a triangle using **p**, **first**, and **t**.

Triangulation consists of traversing the neighbor ring from **first** to **last** using two adjacent points on the neighbor ring at a time. The edges are created in a counterclockwise direction with respect to the surface normal. In the example in Figure 10, the surface normal is pointing out of the page. Edges are drawn from **p** to **first**, **first** to **t**, and **t** to **p**, in that order. Arrows demonstrate the order of the vertices in the triangle. After the triangle is added to the output list the triangulation advances to **t** and **s**, then **s** and **r**, and finally **r** and **last**. Once a triangle is added to the output list it may not be removed.

6.5 Updating the Edge Ring

Once the triangulation is complete, the pivot point is removed from this section of the edge ring. If the pivot point only appeared in the edge ring once (indicated by a status of 1, since the status doubles as an instance counter), the point's status is set to DONE. Otherwise, the status of the pivot point is decremented.

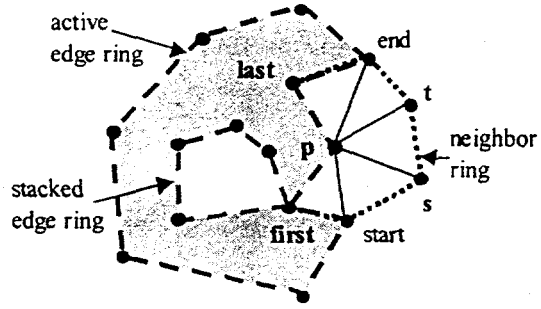

Figure 11: Points removed from edge ring while triangulating **p**.

If **first** and **last** are no longer the start and end points in the neighbor ring, then all of the points before start and after end in the neighbor ring that are duplicated in the edge ring need to be removed from this section of the edge ring. If this place is the only one where a duplicated point appears in the edge ring, as in the example of **last** in Figure 11, then the point's status field is set to DONE. Notice **last** is now surrounded by the triangulated surface, so it cannot be used again. Otherwise, as in the case of **first** in Figure 11, the duplicated point's status is decremented.

Once the duplicates have been removed from the edge ring, the remainder of the points in the neighbor ring, excluding the points pointed to by start and end, are inserted in counterclockwise order into the edge ring. In Figure 11, the remainder of the neighbor ring consists of the two points **s** and **t**. As each of these points is inserted into the edge ring, it is marked as USED.

If a point is encountered that is already USED, the first other instance of the point in the edge ring is located. Depending on the location of the other instance, one of two things will happen. If the other instance is in the active edge ring, the edge ring is split into two rings and one of the rings is pushed onto a stack. In Figure 11, during an earlier point in the triangulation **first** was used for a second time by a point above **first** and the ring was split.

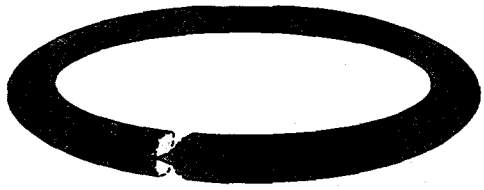

Figure 12: Rejoining the edge ring.

If the other instance is not found in the active edge ring, the stack of edge rings is searched in top down order so that the most recently split edge ring will be found first. Once the other instance of the edge ring point is found in a stacked edge ring, the two edge rings are joined into a single edge ring with the shared edge ring point appearing twice in the same ring. The triangulation of the ring in the hydrogen data set, see Figure 21,

produced this situation as the edge ring was rejoined when the triangulation finished the ring structure, as is shown in Figure 12.

Once the edge ring is finished being updated, the new point is the point that had been the end particle in the neighbor ring.

7 TRIANGULATING A BOUNDARY POINT

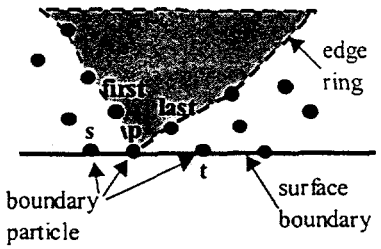

Figure 13: Edge ring intersects surface boundary at **p**.

Boundary points present a problem with respect to the neighbor ring approach presented above. As can be seen in Figure 13, the neighbor ring for point **p** will consist of two disjoint sections on either side of **p**. Even if the algorithm triangulates **p** in parts, it cannot remove **p** from the edge ring. If the algorithm does remove **p**, the removal would either fragment the edge ring or cause nonadjacent points to be adjacent in the edge ring (and hence to be viewed as neighbors by the algorithm which leads to overlapping triangles).

Our solution has two parts. First, we avoid using *boundary* points as pivot points with only a few exceptions. These exceptions include the beginning of a new edge ring where both points are *boundary* points, and when the algorithm is triangulating the areas around surface boundary corners. Otherwise, the algorithm simply advances to the next point in the edge ring. In the example in Figure 13, the algorithm can triangulate the edge to the right of **p** using **last** as the pivot point. As the algorithm circles around the edge ring back towards **p**, it can triangulate the edge area to the left of **p** using **first** as the pivot point. Secondly, the algorithm leaves all the *boundary* points in the edge ring until it makes a complete circuit of the edge ring without any new triangles being added. Then the algorithm marks the *boundary* points as DONE and frees the edge ring so that it can start a new disconnected surface component.

7.1 Edge Ring Consists of Two Boundary Points

When a new edge ring consists of only two *boundary* points, the algorithm must use a *boundary* point as a pivot point. If we were to use our neighbor ring approach without modification, the algorithm would create either a zero area triangle or a triangle outside the surface when it connects the two *boundary* points on either side of the pivot point. An example of this is shown in Figure 14, where **p** and **first** are the only two points on the edge ring.

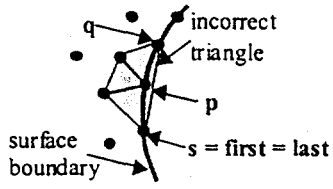

Figure 14: Unmodified neighbor ring produces incorrect triangle.

However, if the algorithm adds **q** to the edge ring and uses it in place of **first**, keeping the other *boundary* point as **last**, then the neighbor ring approach can be used without modification. So prior to calling the neighbor ring routine, the algorithm searches **p**'s neighbors for a third point that is UNUSED and has a boundary edge in common with **p**. Although *interior* neighbors would also avoid the error, the algorithm might select an *interior* point that would be culled from the triangulation, whereas the *boundary* point would not be culled. By not requiring the new point to have a type identical to **p**, the algorithm keeps from eliminating *corner* points.

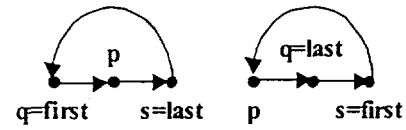

Figure 15: Inserting the third point in the edge ring.

Once the algorithm has found the third point, **q**, it must decide whether to insert **q** before or after **p** in the edge ring. The placement of the third point in the edge ring determines whether the third point is used as **first** or **last** in the neighbor ring, as is shown in Figure 15. In the example in Figure 14, the algorithm needs to insert **q** after **p** so that the range of allowable angles will include points inside the volume boundary. The decision depends on which edge is involved and where the third point is located along the edge relative to **p**.

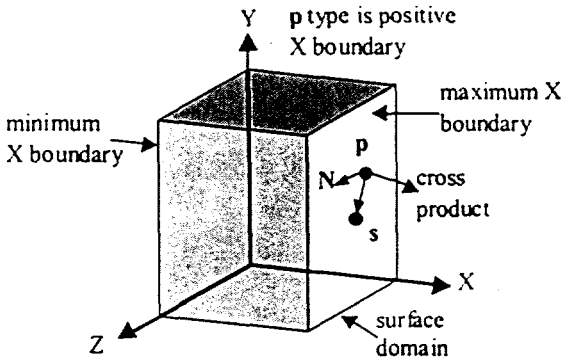

Figure 16: Deciding to insert the third point before or after **p**.

The algorithm takes the cross product of the normal at **p** and a vector between **p** and the second point in the edge ring. The algorithm then compares the direction of the resulting vector with the type of **p**. The third point is inserted before **p** if one of two conditions exists. Either **p** lies on a minimum surface boundary and the corresponding component of the cross product vector is negative, or p lies on a maximum surface boundary and the corresponding component of the cross product vector is positive. Otherwise, the third point is inserted after **p**. Figure 16 illustrates the use of the cross product for the configuration previously given in Figure 14.

7.2 Surface Boundary Corners

The other exception to the policy of not using *boundary* points as pivots, is when the edge ring has adjacent points whose types indicate a transition from one surface boundary to another. In Figure 17, the edge ring transitions from the maximum X boundary to the maximum Y boundary at point **t**. Point **p** is of type *positive x boundary*, **q** is a *positive y boundary* point, and point **t** is both of these types, so it is also a *corner* type. Even if

point **t** were not in the edge ring, the transition in types between **p** and **q** would indicate the transition between surface boundaries.

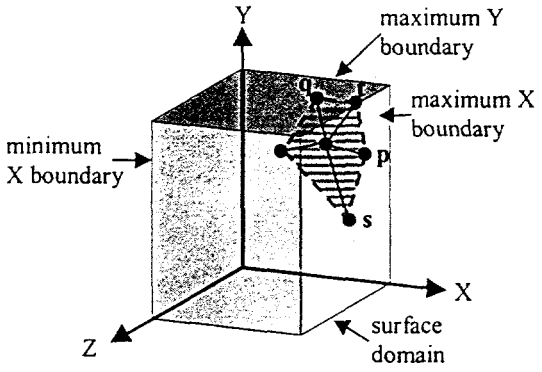

Figure 17: Edge ring transitions between X and Y boundaries.

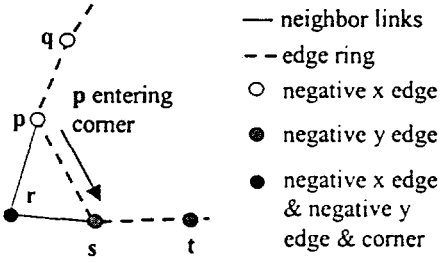

Figure 18: The pivot point is entering a corner.

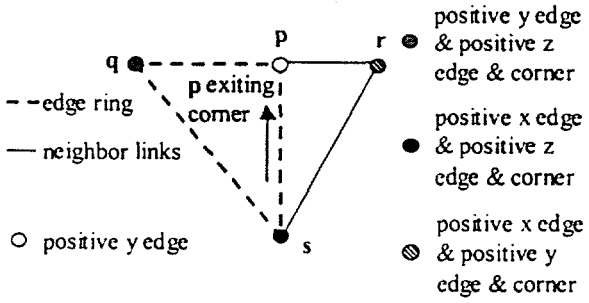

Figure 19: The pivot point is exiting a corner.

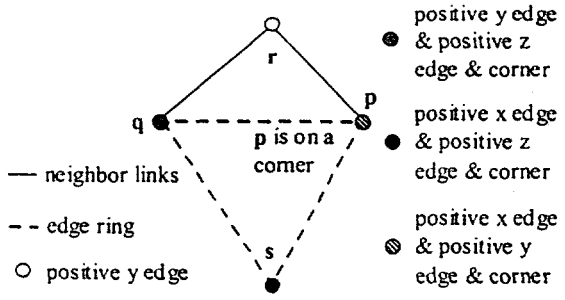

Figure 20: The pivot point is on a corner.

There are three cases requiring special treatment: **p** is entering a corner, **p** is exiting a corner, and **p** is on the corner. Examples of these are shown in Figure 18,

Figure 19, and Figure 20, respectively. In each case, due to the configuration of *boundary* and *corner* points, the point **r** would never be included in the triangulation due to the lack of neighboring *interior* points. Therefore, the algorithm cannot advance to the next point in the edge ring without first triangulating using **p** as the pivot point. The algorithm uses the same approach as before, inserting another UNUSED *boundary* point into the edge ring either before or after **p**, then triangulating using the regular neighbor ring approach. In all of these cases, the UNUSED point inserted into the edge ring is **r**.

8 RESULTS AND FUTURE WORK

We generated point sets using a particle system that isosurfaced various volume data sets. The timings for the SE algorithm to generate triangulations of these point sets are presented in Table 1. The hardware platform was an SGI High Impact Indigo^2 running IRIX 6.3 on an R4400 with 192 megabytes of memory. The number of points input and the number of triangles generated provide a measure of the problem size in each case. Renderings of the triangulated point sets are shown in Figures 21-25.

Point Sets	# Points	# Triangles	Time (secs)
Lobster	42906	83893	2.30
Electron Cloud	3755	7046	.15
Blast Wave	1393	2662	.07
Hydrogen	1516	3020	.08
Hyperboloid	215	366	.01

Table 1: Timings for SE algorithm on various point sets.

In future work, we would like to evaluate the additional time needed to synthesize various inputs that the SE algorithm currently requires. For instance, Hoppe generates normal information as part of his algorithm and computes the *k*-nearest neighbors for each point. Could we find alternate ways to create these inputs that would maintain our speed advantage? Also, once we have the normal and nearest neighbor information, can we determine which points are *interior* and which points are *boundary* points without doing expensive operations like a Delaunay triangulation? It would be useful to develop techniques to fill in whichever pieces are missing in the input and to quantify the costs of using them.

REFERENCES

[1] Amenta, N., M. Bern, and M. Kamvysselis. A New Voronoi-Based Surface Reconstruction Algorithm. In *SIGGRAPH 98 Conference Proceedings, Annual Conference Series*, pages 415-421. ACM SIGGRAPH, Addison Wesley, July 1998.

[2] Crossno, P. and E. Angel. Isosurface Extraction Using Particle Systems. In *Proceedings of Visualization '97*, pages 495-498. IEEE, October 1997.

[3] Edelsbrunner, H. and E. Mucke. Three-Dimensional Alpha Shapes. *ACM Transactions on Graphics*, 13 (1): 43-72, January, 1994.

[4] Hill, F. The Pleasures of "Perp Dot" Products. *Graphics Gems IV*, pages 138-148. AP Professional, Boston, 1994.

[5] Hoppe, H., T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface Reconstruction from Unorganized Points. In *Computer Graphics (SIGGRAPH 92 Conference Proceedings)*, 26 (2): 71-78. Addison Wesley, July 1992.

[6] Teichmann, M. and M. Capps. Surface Reconstruction with Anisotropic Density-Scaled Alpha Shapes. In *Proceedings of Visualization '98*, pages 67-72. IEEE, October 1998.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

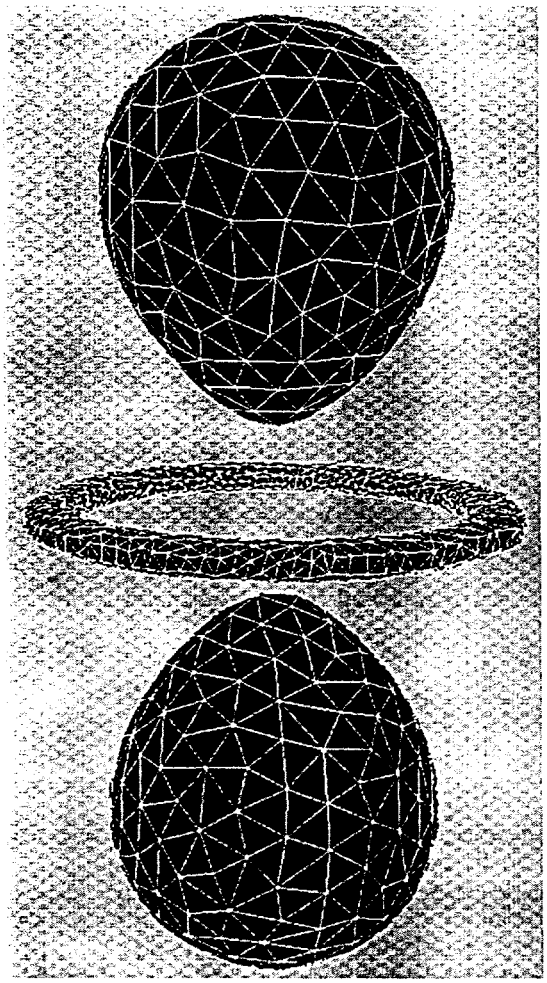

Figure 21: Hydrogen point set.

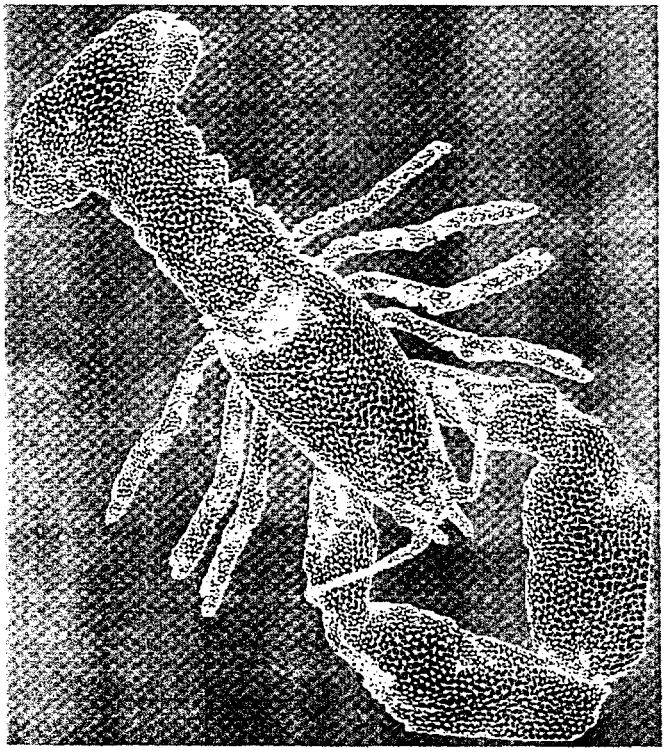

Figure 22: Lobster point set.

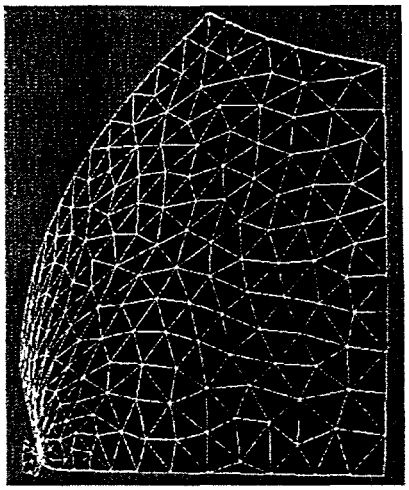

Figure 23: Hyperboloid point set.

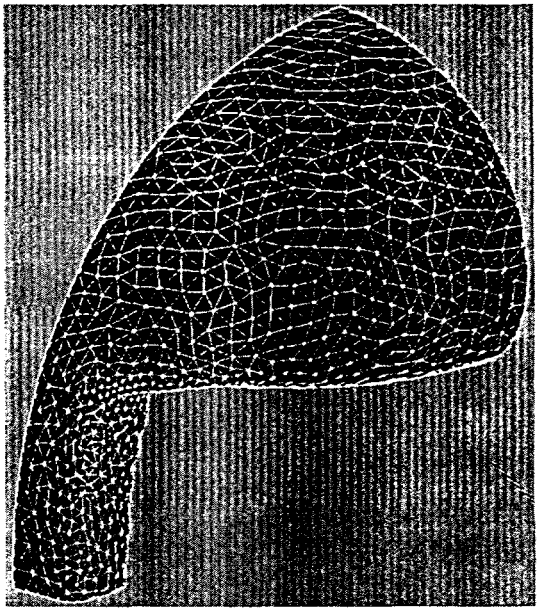

Figure 24: Blast wave point set.

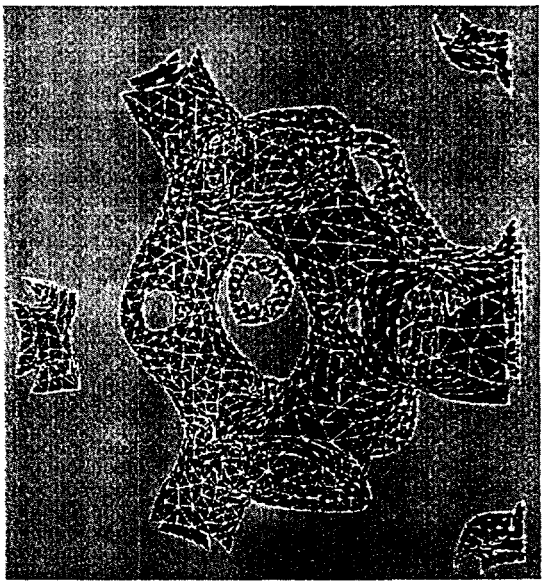

Figure 25: Electron cloud point set.